DISKRETISIERUNG: LINIEN ZEICHNEN

Diskretisierung

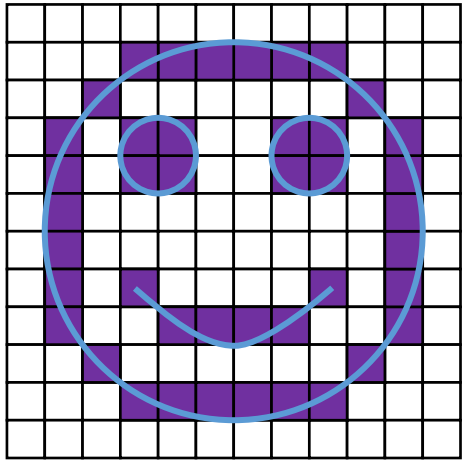
Überführung einer kontinuierlichen Menge Ω in eine diskretisierte, üblicherweise auch endliche, "möglichst passende" Repräsentation M.

Manchmal $M \subset \Omega$.

Diskretisierung: Beispiele

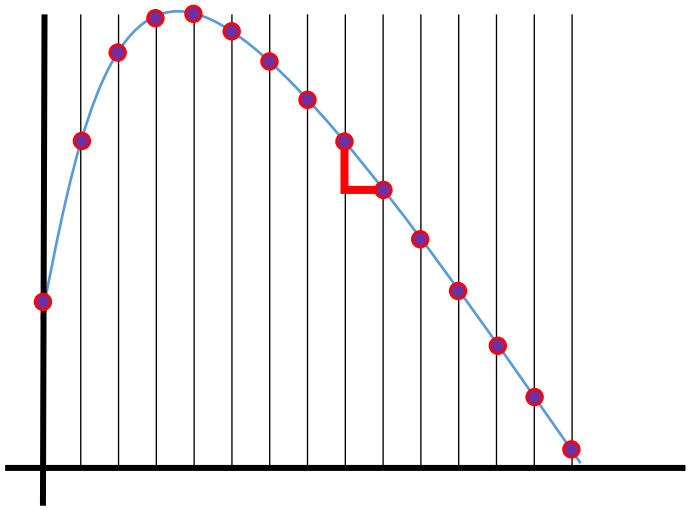
Computergraphik: Abbildungen über $\mathbb{R}^2$ werden zu gefärbten *Pixeln* über einem Rechteck

$$\{0, \dots, w-1\} \times \{0, \dots, h-1\}$$



Numerische Integration: Diskretisierung des Definitionsbereiches

Differentiale werden zu Differenzen



Linien zeichnen

Angenommen Startpunkt p und Enpunkt q einer Linie im $\mathbb{R}^2$ liegen bereits auf dem Gitter, also

$$p = (p_x, p_y) \in \mathbb{Z}^2$$
$$q = (q_x, q_y) \in \mathbb{Z}^2$$

Linie von p nach q im $\mathbb{R}^2$

$$\overline{pq} = \{p + \lambda(p - q) : 0 \leq \lambda \leq 1\} \subset \mathbb{R}^2$$

Entsprechende Linie im $\mathbb{Z}^2$?

Diskrete Linien

Betrachte flache Linie mit

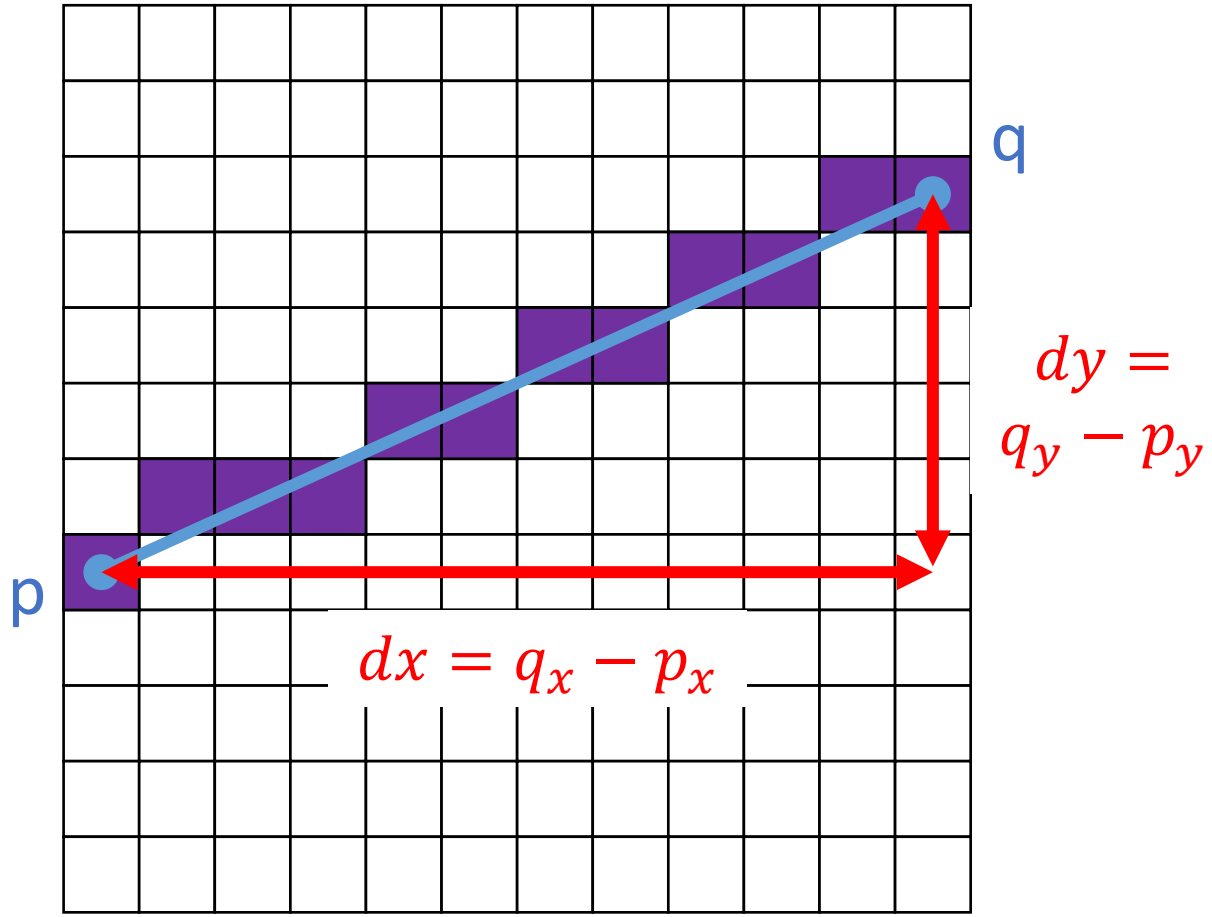
$$|dy| < |dx|$$

Liniengleichung

$$y(x) = p_y + (x - p_x) \cdot \frac{dy}{dx}$$

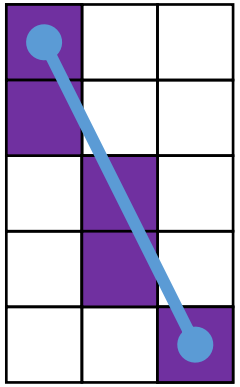
Diskretisiere durch Runden:

$$\hat{y}(x) = p_y + \left\lfloor (x - p_x) \cdot \frac{dy}{dx} + \frac{1}{2} \right\rfloor$$

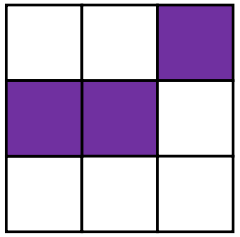


Bemerkungen

Genauso für steile Linien (Tausche Rolle von y und x)

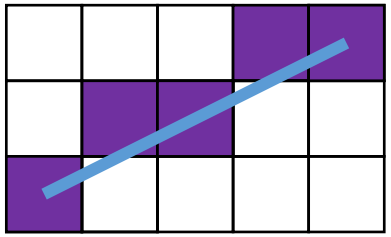


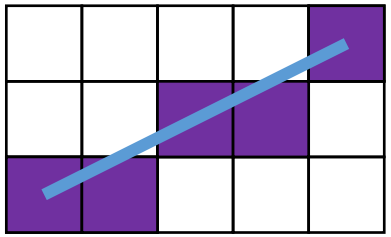
Linien sind 8-verbunden, d.h. benachbarte diskrete Punkte der Linie liegen in der 8er-Nachbarschaft



8 verbunden

Für flache Linien wählt man alle Punkte im Gitter, deren vertikaler Abstand zur Linie maximal 0.5 beträgt.
Bei Gleichheit entweder nur den oberen oder nur den unteren Punkt

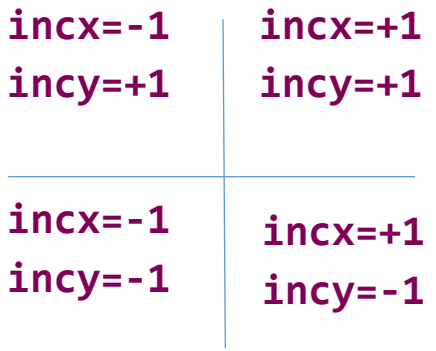


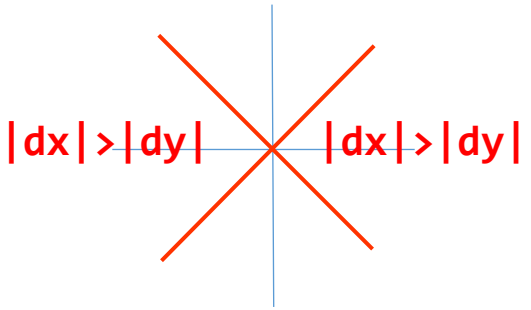


Live Coding: Linien zeichnen

Draw Line Implementation

```
public static void Line(ImageViewer v, int px, int py, int qx, int qy) {
    int dx = (qx-px);
    int dy = (qy-py);
    int incx = 1; if (dx < 0) incx = -1;
    int incy = 1; if (dy < 0) incy = -1;
    if (dx*incx > dy*incy) { // horizontal
        for (int x = px; x <= qx; x+= incx) {
            int y = py + (int)((double)(x-px) * (dy) / dx + 0.5);
            v.dot(x, y);
        }
    } else { // vertical
        for (int y = py; y <= qy; y+= incy) {
            int x = px + (int)((double)(y-py) * (dx) / dy + 0.5);
            v.dot(x, y);
        }
    }
}
```





Fliesskommaarithmetik ➔ Integer Arithmetik

```
int dx = (qx-px);
int dx2 = dx / 2;
...
Ersetze
  int y = py + (int)((double)(x-px) * (dy) / dx + 0.5);
durch
  int y = py + ((x-px)*dy+dx2)/dx;
```

Superkompakt: Bresenham Algorithmus

```
int dx =  Math.abs(x1-x0);
int sx = x0<x1 ? 1 : -1;
int dy = -Math.abs(y1-y0);
int sy = y0<y1 ? 1 : -1;
int err = dx+dy;

while(x0 != x1 || y0 != y1) {
   dot(x0, y0);
   if (2*err > dy) {
      err += dy;
      x0 += sx;
   }
   if (2*err < dx) {
      err += dx;
      y0 += sy;
   }
}
dot(x1,y1);
```

Idee des Algorithums:

Geradengleichung im $\mathbb{R}^2$:

$$dy \cdot (x - p_x) - dx \cdot (y - p_y) = 0$$

Ständige Korrektur des Diskretisierungsfehlers

$$\varepsilon = dy \cdot (x - p_x) - dx \cdot (y - p_y)$$

während des Durchlaufens der Geraden.

Wir vertiefen das hier nicht.

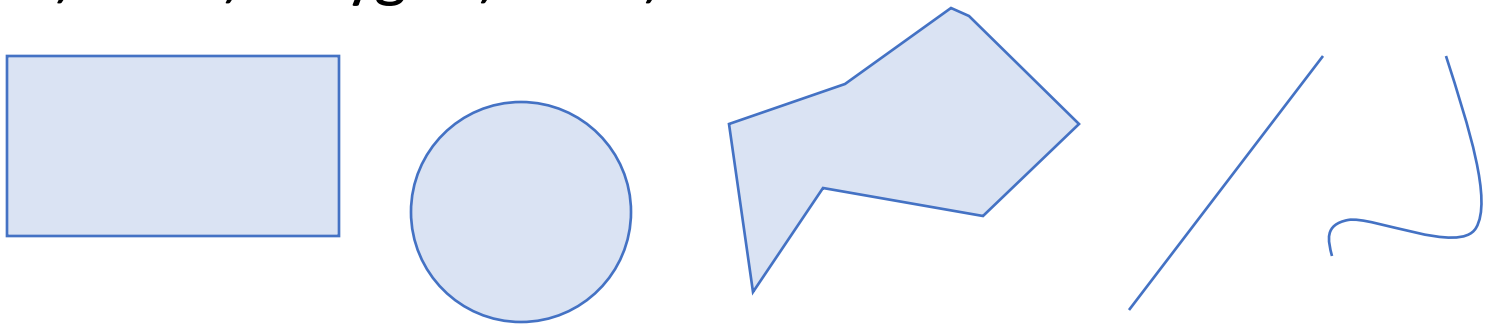
Motivation Zeichenprogramm, Vererbung, Polymorphie, Fallstudie Numerische Integration: Rechteck-, Trapez- und Simpson-Regel, Monte-Carle Integration

6. OBJEKTORIENTIERTE PROGRAMMIERUNG

Vererbung – Motivation

Beispielhaftes Ziel: Grafikbibliothek mit erweitertem Fundus an geometrischen Figuren

- Rechteck, Kreis, Polygon, Linie, Kurve etc.



Gemeinsame Eigenschaften

- Randfarbe, Randdicke, Füllfarbe, Muster, Offset, etc.

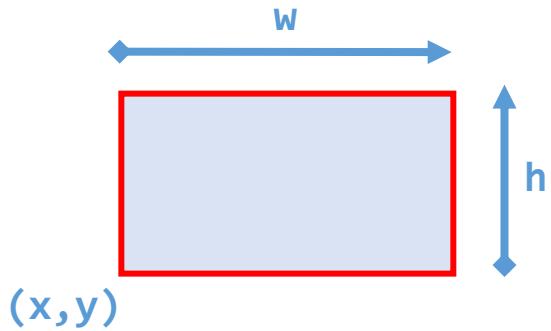
Gemeinsame Operationen

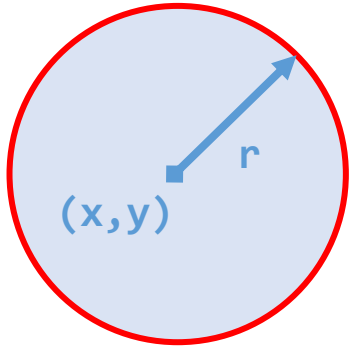
- Zeichnen, Füllen, PointInX, Schneiden mit Geraden, etc.

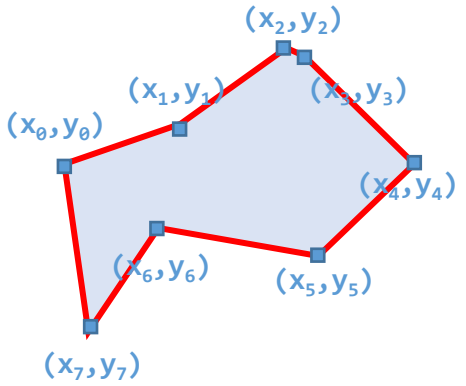
Vererbung – Motivation

Unterschiede der Figuren:

- **Repräsentation**
  - Rechteck: x,y,w,h,
  - Kreis: x,y,r,
  - Polygon: Liste von Eckpunkten
- **Implementation der Operationen**
  - Zeichnen / Füllen verschieden aufwändig
  - Schnitt mit Gerade oder Erkennung eines innenliegenden Punktes anders zu implementieren
  - etc.







Vererbung – Motivation

Annahme: wir wollen ein Zeichenprogramm schreiben, welches z.B. über eine (verkettete) Liste auf seine geometrischen Objekte zugreift.

Wie lässt sich das realisieren?

Eine Möglichkeit (noch ohne OOP):

- Packe alle nötige Information für ein geometrisches Objekt in dieselbe Datenstruktur
- Füge eine «Schaltervariable» hinzu, mit der über die Art («den Typ») des Objektes entschieden werden kann.

Motivation (noch keine Vererbung)

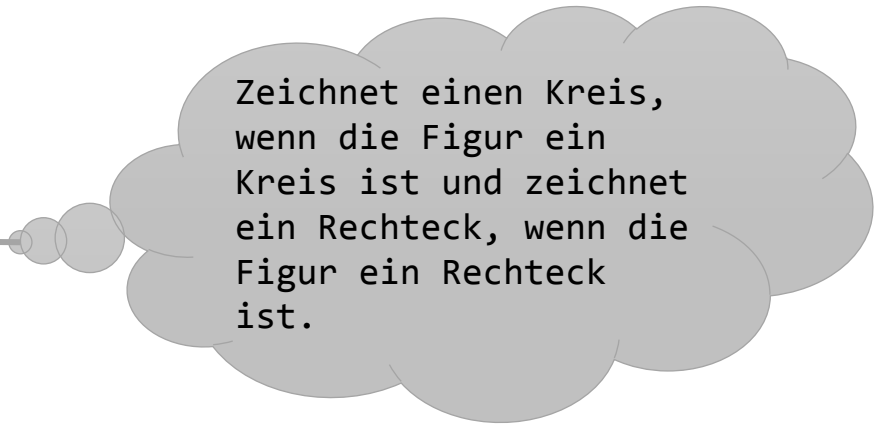
```
public class Figure {
   Figure next;    //  für Liste
   int typ;        // 0: Rechteck, 1: Kreis
   Color col;      // Farbe
   int x,y,w,h,r;  // Parameter für jede mögliche Figur (Kreis / Rechteck)

   public Figure(int Typ, Color Col, int X, int Y, int W, int H, int R)
   {
      x = X; y = Y; w = W; h = H; r = R; col = Col;
      typ = Typ;
   }
   ...
}
```

Motivation (noch keine Vererbung)

```
public class Figure {
   ...
   void DrawCircle(BufferedImage img) { … }
   void DrawRectangle(BufferedImage img) { … }

   public void Draw(ImageViewer v) {
      v.setColor(col);
      if (typ == 0)
         DrawRectangle(v);
      else
         DrawCircle(v);
      v.repaint();
   }
}
```

Eine Liste von Figuren

```
public class FigureList {
   Figure first = null, last = null;

   public void Add(Figure f) {
      if (last == null) first = f;
      else last.setNext(f);
      last = f;
   }

   public void Draw(ImageViewer v){
      for (Figure f = first; f != null; f = f.next)
         f.Draw(v);
   }
}
```



Vererbung – Motivation

Nachteile obigen Vorgehens?

- Abundante Information

  z.B. `w,h` nicht genutzt für den Kreis

  z.B. `DrawCircle` nicht verwendet für Rechteck

- Administrativer Overhead

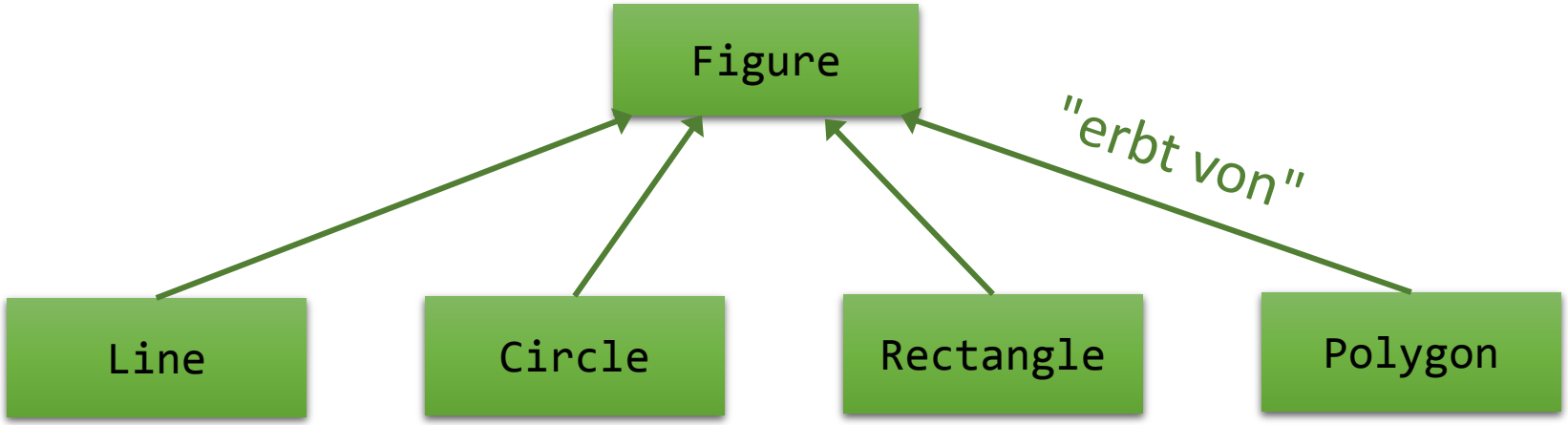
  Der Programmierer muss Fallunterscheidung programmieren

- Nicht-invasives Erweitern des Codes unmöglich.

  Möchte man einen neuen Typ von Figur hinzufügen, so muss man den ursprünglichen Code verändern.

Vererbung – Motivation

Idee zur Vermeidung obiger Nachteile: Darstellung jeder geometrischen Figur als **Erweiterung** eines zugrunde liegenden **Grundtyps** `Figure`



Gemeinsame Eigenschaften verbleiben (nur) in der Basisklasse Shape. Erklärung folgt.

Klassen können Eigenschaften *(ver)erben*

```
public class Figure {
   Color col = Color.black;
   public Figure next = null;

   public void setNext(Figure nxt){
      next = nxt;
   }

   public void setColor(Color C){
      col = C;
   }
}
```

Datenfelder, welche für alle Figuren deklariert sind

Methoden, welche für alle Figuren deklariert sind

Klassen können Eigenschaften *(ver)erben*

```
public class Circle extends Figure {
   int x,y,r;

   // Konstruktor eines Kreises
   public Circle(Color c, int X, int Y, int R) {
      setColor(c);
      x = X; y = Y; r = R;
   }

   ...
}
```

Datenfelder, welche nur für `Circle` deklariert sind

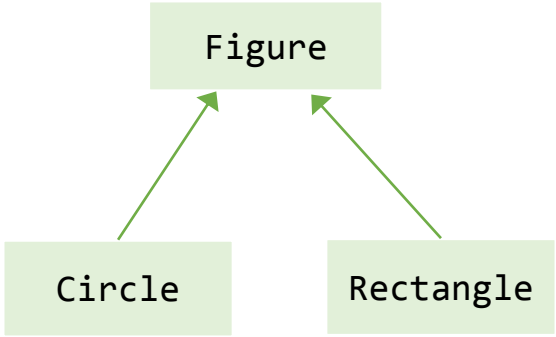
Klassen können Eigenschaften *(ver)erben*

```
public class Rectangle extends Figure {
   int x,y,w,h;

   // Konstruktor eines Rechtecks
   Rectangle(Color c, int X, int Y, int W, int H) {
      setColor(c);
      x = X; y = Y; w = W; h = H;
   }
   ...
}
```

Datenfelder, welche nur für `Rectangle` deklariert sind

Klassen können Eigenschaften *(ver)erben*

```
public class Figure { ... }
public class Circle extends Figure { ... }
public class Rectangle extends Figure { ... }
```




```
...
Rectangle rectangle=new Rectangle(100,100,200,100);
Circle circle= new Circle(100,100,50);
circle.SetColor(Color.blue);
rectangle.SetColor(Color.red);
...
```

Aufruf der Methoden der Basisklasse `Figure`

➔ Wiederverwendbarkeit gemeinsam nutzbaren Codes durch ***Generalisierung***

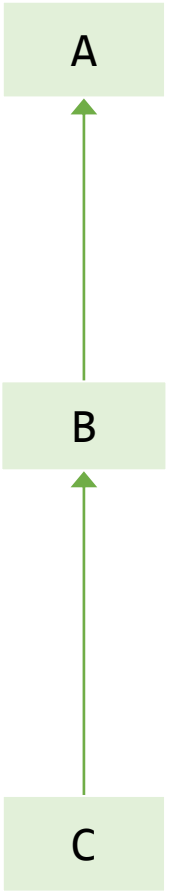
Vererbung – Nomenklatur

```
class A {
      ...
}

class B extends A {
      ...
}

class C extends B {
      ...
}
```

Basisklasse
(Superklasse)

Abgeleitete Klasse
(Subklasse)

«B und C *erben von* A»
«C *erbt von* B»



Vererbung – Kompatibilität

Ein abgeleitetes Objekt kann überall dort verwendet werden, wo ein Basisobjekt gefordert ist, aber nicht umgekehrt.

```
Rectangle rectangle = new Rectangle(0,0,10,10);
Figure figure = rectangle;  // ok: Rectangle extends Figure
figure.setColor(Color.red); // ok: Rectangle verwendet eine Methode von seiner Basisklasse
Rectangle r = new Circle(0,0,10);  // type mismatch: cannot convert from Circle to Rectangle
Figure c = new Circle(0,0,10);  // ok: Circle extends Figure
Circle circle = c; // type mismatch: cannot convert from Figure to Circle
```

Vererbung – Sichtbarkeit

In der abgeleiteten Klasse können Variablen und Methoden der Basisklasse direkt verwendet werden

Voraussetzung:
Sichtbarkeit gewährleistet

Sichtbarkeiten nach Modifizierer

Modifizierer	Klasse	Paket	Sub-Klasse	Global
public	✓	✓	✓	✓
protected	✓	✓	✓	
(keiner)	✓	✓		
private	✓			

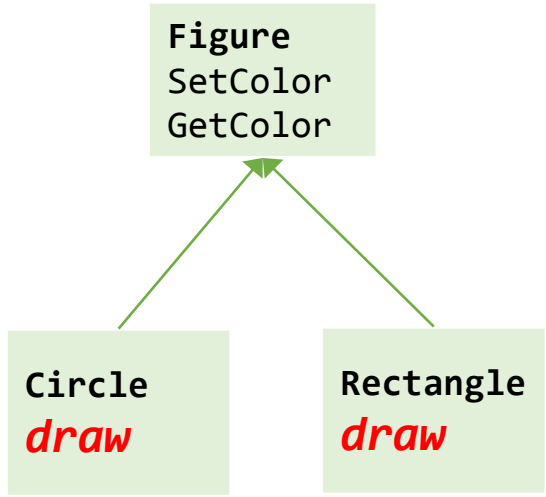
```
public class Rectangle extends Figure {
  ...
  // zusätzlicher Konstruktor
  Rectangle (Color col; int rx, int ry, int rw, rh) {
    x = rx; y = ry; w = rw; h = rh;
        SetColor(col);
  }
  ...
}
```

ruft die Methode SetColor in der Basisklasse `figure` auf

Motivation – noch ohne Polymorphie

Ziel: Zeichnen der Figur entsprechend Ihrer Eigenschaften durch Anbringen einer entsprechenden Methode, z.B. bei Kreis und Rechteck:

```
public class Circle extends Figure {
   ...
     public void Draw(ImageViewer v) { ... } // Zeichne Kreis
}

public class Rectangle extends Figure {
   ...
     public void Draw(ImageViewer v) { ... } // Zeichne Rechteck
}
```



Motivation – noch ohne Polymorphie

Ziel: Zeichnen der Figur entsprechend Ihrer Eigenschaften. Nun kann die jeweilige Methode draw aufgerufen werden

```
ImageViewer v = new ImageViewer(400,400);
Rectangle rectangle=new Rectangle(100,100,200,100);
Circle circle= new Circle(100,100,50);
circle.SetColor(Color.blue);
rectangle.Draw(v);
circle.Draw(v);
v.update();
```

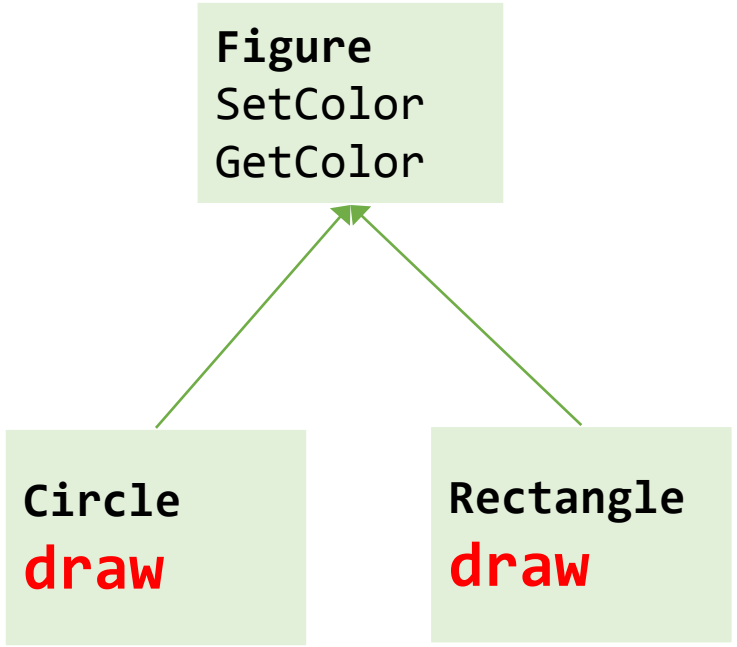
Ruft die jeweilge draw Methode auf, gemäss **statischem** Typ.

Ausprobieren!

```
public class FigureList {
    Figure first = null, last = null;

    public void Add(Figure f){
        if (last == null) first = f;
        else last.setNext(f);
        last = f;
    }

    public void Draw(ImageViewer v) {
        for (Figure f = first; f != null; f = f.next)
            f.Draw(v);
    }
}
```

"The method Draw(ImageViewer) is undefined for the type Figure"

Erklärung




```
Circle circle = new Circle(…);
circle.Draw(img); // ok
Figure figure = circle;
figure.Draw(img); // Fehler: Methode draw bei
                          figure nicht definiert
```

`figure` hat statischen Typ `Figure`, der hat kein `draw`.

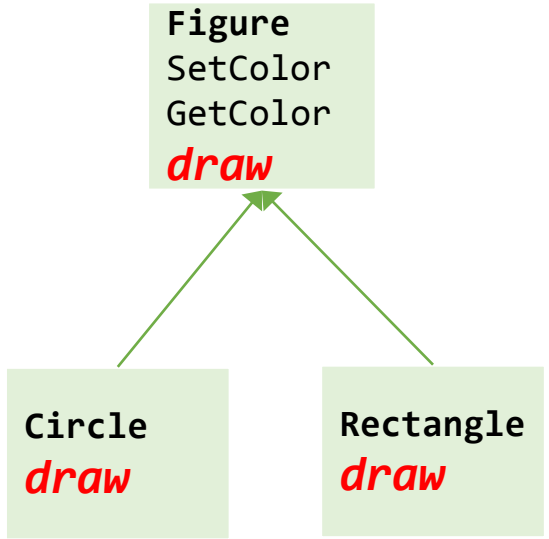
Die Magie der Polymorphie

Polymorphie (in Java): Deklariert man eine (nicht statische) Methode mit gleichem Namen und gleicher Signatur in abgeleiteten Klassen *und* in der gemeinsamen Basisklasse, so wird *zur Laufzeit* automatisch über die auszuführende Variante entschieden.

Man sagt, die Methode wird in der abgeleiteten Klasse *überschrieben*.

Polymorphie in Java

```
public class Figure {
   ...
   public void draw(BufferenImage img) { }
}

public class Circle extends Figure {
   ...
   public void draw(BufferedImage img) {...} // draw circle
}

public class Rectangle extends Figure {
   ...
   public void draw(BufferedImage img) {   ... } // draw rect
}
```



Polymorphie

Bei überschriebenen Methoden wird der *dynamische* Typ gewählt («dynamic dispatching»)

```
ImageViewer v = new ImageViewer(400,400);

Figure f1 = new Rectangle(100,100,200,100);
Figure f2 = new Circle(100,100,50);
f1.draw(v); // wählt automagisch draw von Rectangle
f2.draw(v); // wählt automagisch draw von Circle

v.repaint();
```

Ziel erreicht!

```
public class FigureList {
   Figure first = null, last = null;

   public void Add(Figure f){
      if (last == null) first = f;
      else last.setNext(f);
      last = f;
   }

   public void Draw(ImageViewer v) {
      for (Figure f = first; f != null; f = f.next)
         f.Draw(v);
   }
}
```

```
FigureList list = new FigureList();
list.Add(new Rectangle(50,50,200,150);
list.Add(new Circle(50,50,50));
list.Add(new Rectangle(0,0,50,50));
list.Draw(v);
```